CS3025 Compiladores

Semana 5:
ASTs y el Visitor Pattern
14 Septiembre 2023

Igor Siveroni

Resumen

- Uso de arboles sintácticos abstractos / abstract syntax trees (ASTs)

- The Visitor Pattern
  Creación de visitantes para implementar las siguientes fases del compilador

- Implementación de Visitantes.

- Lectura:
  Capitulo 4: Abstract Syntax
  Modern Compiler Implementation. Andrew Appel.

Arboles de Sintaxis Abstractos (AST) – Sintaxis Abstracta

Habíamos visto que, luego del análisis sintáctico, la manera mas eficiente y clara de comunicar la estructura del programa original (el programa a compilar) a los fases siguientes del compilador es mediante Arboles de Sintaxis Abstracta (ASTs).

El compilador, en las siguientes fases, puede hacer uso de otras Representaciones intermedias (IR: Intermediate Representations).

Los ASTs reflejan una gramática que sintetiza o abstrae la gramática original. A esta nueva gramática se le llama Gramática Abstracta.

Los ASTs son construidos mediante acciones semánticas asociadas a no-terminales y producciones analizadas por el analizador sintactico (parser).

El compilador necesitara representar y manipular ASTs utilizando estructuras de datos.
Estrategia: Una clase abstracta por no-terminal y una subclase por cada producción.

Arboles de Sintaxis Abstractos (AST) – Sintaxis Abstracta

Por ejemplo, la sintaxis modificada que hace posible el analisis sintactico de expresiones aritméticas usando descenso recursivo es:

```
Exp ::= Term ((+|-) Term)*
        Term ::= Factor ((*|/) Factor)*
        Factor ::= num | '(' Exp ')'
```

Pero, una vez resueltos asociatividad y order de precedencia de operadores, solo es necesaria una gramática o sintaxis abstracta de la siguiente forma:

```
Exp ::= Exp (+|-|*|/) Exp |
        num |
        '(' Exp ')'
```

En base a la gramática abstracta, se definira una clase abstracta por no-terminal y una subclase por cada producción.

Arboles de Sintaxis Abstractos (AST) – Implementacion

Siguiendo con el ejemplo de las expresiones aritméticas, los ASTs generados por la sintaxis

```
Exp ::= Exp (+|-|*|/) Exp | num |  '(' Exp ')'
```

son implementados por las siguientes clases:

```
enum BinaryOp { PLUS, MINUS, MULT,
DIV, EXP};

class Exp { …  virtual ~Exp() = 0;};

class BinaryExp : public Exp {
public:
  Exp *left, *right;
  BinaryOp op;
  BinaryExp(Exp* l, Exp* r, BinaryOp
op);
```

```
class NumberExp : public Exp {
public:
  int value;
  NumberExp(int v);
… };

class ParenthExp : public Exp {
public:
  Exp *e;
  ParenthExp(Exp *e);
… };
```

Arboles de Sintaxis Abstractos (AST) – Recorrido / Visitas

Las siguientes fases del compilador se realizan via recorridos de los nodos internos y hojas del AST. Si, por ejemplo, queremos evaluar e imprimir el árbol, debemos declarar las funciones correspondientes en la clase abstracta

```
class Exp {
public:
  virtual void print() = 0;
  virtual int eval() = 0;
  static string binopToString(BinaryOp op);
  virtual ~Exp() = 0;
};
```

e implementar los métodos en cada subclase. Las implementaciones de eval() para `NumberExp` y `ParenthExp` son :

```
int NumberExp::eval() {
  return value;
}

int ParenthExp::eval() {
  return e->eval();
}
```

Arboles de Sintaxis Abstractos (AST) – Recorrido / Visitas

Y para BinaryExp,

```
int BinaryExp::eval() {
  int v1 = left->eval();
  int v2 = right->eval();
  int result = 0;
  switch(this->op) {
  case PLUS: result = v1+v2; break;
  case MINUS: result = v1-v2; break;
  case MULT: result = v1 * v2; break;
  case DIV: result = v1 / v2; break;
  }
  return result;
}
```

Arboles de Sintaxis Abstractos (AST) – Recorrido / Visitas

- Esto implica que, cada vez que tenemos que implementar una nuevo análisis o fase del compilador – llamémosles **interpretaciones** - tenemos que:
  - Crear una nueva función en la clase abstracta
  - Implementar la función en todas las subclases

- Esto no es escalable, todas las funciones estarían asociadas explícitamente a las subclases del AST e implementadas en un solo archivo con todos los análisis juntos (si implementamos las clases del AST en un solo archivo) o repartidas en varios archivos, uno por subclase.

- Lo que queremos es un archivo / objeto por análisis / fase del compilador.

Necesitamos implementar el Visitor Pattern

Visitor Pattern

- La implementación actual sigue el estilo *orientado a objetos*.
- Para el tipo de problema que estamos atacando buscamos un estilo de *sintaxis separada de la interpretacion*, donde nuevas interpretaciones e.g print, eval; pueden ser añadidas con facilidad.
- Para esto usaremos una técnica llamada **Visitor Pattern**

Visitor Pattern / Patron Visitor:

- Un visitor implementa una sola interpretación – tiene **una sola tarea a ejecutar**.
- El visitor es un objeto que contiene un método `visit` por cada clase del AST.
- Cada clase del AST debe contener un método accept: El método acepta al visitor y pasa el control de regreso al método correspondiente en el visitor.
- De este modo, el control va ida y vuelta entre el visitor y las clases del AST.
- Cada visitor implementa (o es subclase de) una interface `Visitor` (una clase abstracta en C++)
- Cada `accept` método toma un `Visitor` como argumento y cada método `visit` tomada un node del AST como argumento.

Visitor Pattern: Ejemplo Visitor para ASTs de Exp

Siguiendo con el ejemplo de las expresiones aritméticas con ASTs generados por la sintaxis abstracta

```
Exp ::= Exp (+|-|*|/) Exp | num |  '(' Exp ')'
```

y clases AST `BinaryExp`, `NumExp` y `ParethExp`, definimos la clase abstracta (interface) de los visitors para esta esta sintaxis de la siguiente manera:

```
class ASTVisitor {
public:
  virtual int visit(BinaryExp* e) = 0;
  virtual int visit(NumberExp* e) = 0;
  virtual int visit(ParenthExp* e) = 0;
};
```

Visitor Pattern: Ejemplo AST para Exp

Las declaraciones de las clases del AST se ven ahora asi:

```
class Exp {
public:
  virtual int accept(ASTVisitor* v) = 0;
  static string binopToString(BinaryOp op);
  virtual ~Exp() = 0;
};

class BinaryExp : public Exp {
public:
  Exp *left, *right;
  BinaryOp op;
  BinaryExp(Exp* l, Exp* r, BinaryOp op);
  int accept(ASTVisitor* v);
  ~BinaryExp();
};
```

```
class NumberExp : public Exp {
public:
  int value;
  NumberExp(int v);
  int accept(ASTVisitor* v);
  ~NumberExp();
};

class ParenthExp : public Exp {
public:
  Exp *e;
  ParenthExp(Exp *e);
  int accept(ASTVisitor* v);
  ~ParenthExp();
};
```

Solo estructura / sintaxis y métodos `accept`

Visitor Pattern: Ejemplo AST para Exp

La implementación de las clases del AST se ven ahora asi:

// constructores

// destructores

```
int BinaryExp::accept(ASTVisitor* v) {
  return v->visit(this);
}

int NumberExp::accept(ASTVisitor* v) {
  return v->visit(this);
}

int ParenthExp::accept(ASTVisitor* v) {
  return v->visit(this);
}
```

Visitor Pattern: Ejemplo AST para Exp

La implementación de un nuevo visitor implica, primero, declarar la nueva clase:

```
class ASTEvaluator : public ASTVisitor {
public:
  int eval(Exp*);
  int visit(BinaryExp* e);
  int visit(NumberExp* e);
  int visit(ParenthExp* e);
};
```

El metodo eval es el punto de entrada a la evaluación:

```
int ASTEvaluator::eval(Exp* e) {
  return e->accept(this);
}
```

Visitor Pattern: Ejemplo AST para Exp

Para `NumExp` y `ParenthExp` (antes y después):

```
int NumberExp::eval() {
  return value;
}

int ParenthExp::eval() {
  return e->eval();
}
```

```
int ASTEvaluator::visit(NumberExp* e)
{
  return e->value;
}

int ASTEvaluator::visit(ParenthExp*
ep) {
  return ep->e->accept(this);
}
```

Visitor Pattern: Ejemplo AST para Exp

Para `BinaryExp` (antes y después):

```
int BinaryExp::eval() {
  int v1 = left->eval();
  int v2 = right->eval();
  int result = 0;
  switch(this->op) {
  case PLUS: result = v1+v2; break;
  case MINUS: result = v1-v2; break;
  case MULT: result = v1 * v2; break;
  case DIV: result = v1 / v2; break;
  }
  return result;
}
```

```
int ASTEvaluator::visit(BinaryExp* e) {
  int v1 = e->left->accept(this);
  int v2 = e->right->accept(this);
  int result = 0;
  switch(e->op) {
  case PLUS: result = v1+v2; break;
  case MINUS: result = v1-v2; break;
  case MULT: result = v1 * v2; break;
  case DIV: result = v1 / v2; break;
  }
  return result;
}
```

Visitor Pattern: Ejemplo AST para Exp

Y, finalmente

```
Exp* e = …

ASTEvaluator evaluator;

cout << "eval " << evaluator.eval(exp) << endl;
```